# 配置自动编译vscode

|  |
| --- |
| 1、npm install -g typescript或者 yarn global add typescript  2. tsc --init 生成tsconfig.json 改 "outDir": "./js",  3、终端 -> 运行任务或ctlr+shift+b 监视tsconfig.json    4、 <script src="js/test.js"></script><!--自行添加js引入-->  -----------------------------------------------------------------------  tsc filename.ts 生成js  tsc filename.ts -d 生成声明文件filename.d.ts 代码提示功能 |

# 基础类型

|  |
| --- |
| typescript中为了使编写的代码更规范，更有利于维护，增加了类型校验，  在typescript中主要给我们提供了以下数据类型  typescript中为了使编写的代码更规范，更有利于维护，增加了类型校验  写ts代码必须指定类型 |

## 布尔类型(boolean)

|  |
| --- |
| *let* flag:*boolean* = true  flag = false  //正确 |

## 数字类型(number)

|  |
| --- |
| *let* num:*number* = 123  num = 456 |

## 字符串类型(string)

|  |
| --- |
| *let* str:*string* = 'hello'  str = 'world' |

## 数组类型(array)

|  |
| --- |
| // 第一种  *let* arr:*number*[] = [11, 22, 33]  // 第二种  *let* arr:*Array*<*number*> = [11, 22, 33]  // 第三种  *let* arr:*any*[] = ['11', 22, true] |

## 元组类型(tuple)

|  |
| --- |
| // 元组类型(tuple)  属于数组的一种  *let* arr:[*number*, *string*]=[11,'22', 33] |

## 枚举类型(enum)

|  |
| --- |
| // 1.数字枚举  // 枚举下标默认从0开始，赋值，则后续值递增  *enum* Color {      blue,      red = 10,      orange  }  *let* b:Color = Color.blue  *let* r:Color = Color.red  *let* o:Color = Color.orange  console.log(b) // 0  console.log(r) // 10  console.log(o) // 11  // 2.字符串枚举  // 在一个字符串枚举里，每个成员都必须用字符串字面量，或另外一个字符串枚举成员进行初始化  *enum* Direction {      NORTH = 'NORTH',      SOUTH = 'SOUTH',      EAST = 'EAST',      WEST = 'WEST'  }  // 3.异构枚举, 异构枚举的成员值是数字和字符串的混合  *enum* Enum {      A,      B,      C = "C",      D = "D",      E = 8,      F,  }  // 数字枚举相对字符串枚举多了 “反向映射”  console.log(Enum.A) //输出：0  console.log(Enum[0]) // 输出：A |

## 任意类型(any)

|  |
| --- |
| *let* num:*any* = 10  num = 'str'  num = true  //任意类型的用处  *let* oBox:*any* = document.getElementById('box')  oBox.style.color = 'red' |

## null和undefined

|  |
| --- |
| 默认情况下 null 和 undefined 是所有类型的子类型。  就是说你可以把 null 和 undefined 赋值给 number 类型的变量。  然而，如果你指定了--strictNullChecks 标记，null 和 undefined 只能赋值给 void 和它们各自的类型  // 定义没赋值就是undefined  // let num:number  console.log(num)  // undefined 报错  // let num:undefined  console.log(num) // undefined 正确  //可能是 number null undefined类型  *let* num:*number* | *null* | *undefined*  num = 10  console.log(num) // 10 |

## void类型

|  |
| --- |
| void表示没有任何类型，一般用于方法没有返回任何类型  *function* func():*void*{      console.log('hello')  }  //错误写法  *function* func(): *undefined*{      console.log('hello')  } |

## never类型

|  |
| --- |
| // never类型:是其他类型(包括null和undefined)的子类型，代表从不会出现的值  //这意味着声明never的变量只能被never类型所赋值  *let* foo:*never*  // foo =123 //错误写法  foo = (() *=>* {      throw new *Error*('错误')  })() |

# 函数

## 函数的定义

|  |
| --- |
| // es5函数声明法  *function* func(*name*) {      return 'func'  }  // 匿名函数  *var* func = *function*(*name*) {      return 'func'  }  //没有返回值的方法  *function* func() {      console.log('func')  }  ===========================================================  // ts中定义函数的方法  //函数声明法  *function* func(*name*:*string*):*string* {      return 'func'  }  // 匿名函数  *let* func = *function*(*name*:*string*):*string* {      return 'func'  }  //没有返回值的方法  *function* func():*void* {      console.log('func')  } |

## 可选参数

|  |
| --- |
| // es5方法的实参和形参可以不一样，  // ts中必须一样，不一样就需要配置可选参数  // 注意:可选参数必须配置到参数的最后面  *function* func(*name*:*string*, *age*?:*number*):*string* {      return *age* ? `${*name*} ${*age*}` : *name*  }  func('zhangsan') // zhangsan  func('zhangsan', 20) // zhangsan 20 |

## 默认参数

|  |
| --- |
| // es5不能设置默认参数  // es6和ts中都可以设置默认参数  *function* func(*name*:*string*, *age*:*number*=20):*string* {     return *age* ? `${*name*} ${*age*}` : *name*  }  func('zhangsan', 20) // zhangsan 20 |

## 剩余参数 也叫可变参数

|  |
| --- |
| // 三点运算符 接受新参传过来的值  *function* func(*a*:*number*, *b*:*number*, ...*result*:*number*[]):*number* {  *let* sum = *a* + *b*      for(*let* i=0; i<*result*.length; i++) {          sum += *result*[i]      }      return sum  }  console.log(func(1,2,3,4,5,6)) // 21 |

## 函数重载

|  |
| --- |
| //es5中出现同名方法，下面会替换上面的方法  *function* func(*a*) {...}  *function* func(*a*, *b*) {...}  // ts重载: 函数名相同，参数类型或参数个数不一样  // 方法是为同一个函数提供多个函数类型定义来进行函数重载  *function* func(*name*:*string*):*string*  function func(*name*:*string*, *age*:*number*):*string*  function func(*name*:*any*, *age*?:*any*):*any* {      return *age* ? `${*name*} ${*age*}` : *name*  }  console.log(func('zhangsan'))  // 正确  console.log(func('zhangsan', 20)) // 正确  console.log(func(123))  // 错误 |

## 箭头函数

|  |
| --- |
| // es5  setTimeout(*function*() {      console.log(this) // 指向window  }, 1000)  *let* obj = {   name: '张三',      func: *function*() {          setTimeout(*function*() {              console.log(this) // 指向window          }, 1000)      }  }  console.log(obj.func())  ===============================================================  // es6 注意：箭头函数里面的this指向上下文  setTimeout(() *=>* {      console.log(this) // 指向window  }, 1000)  *let* obj = {      name: '张三',      func: *function*() {          setTimeout(() *=>* {              console.log(this) // 指向obj          }, 1000)      }  }  console.log(obj.func()) |

# ES5类、对象、继承

## 构造函数和原型链增加方法

|  |
| --- |
| *function* Person(*name*){    // 构造函数Person里的属性、实例方法不会被多个实例共享    this.name = *name*  // 实例方法    this.run = *function*() {        console.log(this.name + '...run')    }  }  //原型链上面的属性、方法会被多个实例共享  Person.prototype.sex = '男'  Person.prototype.work = *function*() {    console.log(this.name + '...work')  }  *let* p1 = new Person('张三')  *let* p2 = new Person('李四')  console.log(p1)  console.log(p2)  console.log(p1.name === p2.name) // false  console.log(p1.run === p2.run) // false  console.log(p1.sex === p2.sex) // true  console.log(p1.work === p2.work) // true |

## 类的静态方法

|  |
| --- |
| *function* Person(){    // 属性    this.name = '张三'    // 实例方法    this.run = *function*() {      console.log(this.name + '...run')    }  }  Person.getInfo = *function*() {    console.log('静态方法')  }  Person.prototype.sex = '男'  Person.prototype.work = *function*(){    console.log(this.name + '...work')  }  *let* p = new Person()  // 调用实例方法  p.work()  //调用静态方法  Person.getInfo() |

## 对象冒充实现继承

|  |
| --- |
| 问题：实例化子类不能继承原型链上的属性和方法  *function* Person() {    this.name = '张三'    this.run = *function* () {      console.log(this.name + '...run')    }  }  Person.prototype.sex = '男'  Person.prototype.work = *function* () {    console.log(this.name + '...work')  }  // 对象冒充实现继承  *function* Student() {    Person.call(this)  }  *let* student = new Student()  // 对象冒充实现继承,可以继承构造函数里面的属性和实例方法  student.run() // 张三...run  // 但实例化子类不能继承原型链上的属性和方法  student.work() // 报错 work is not a function |

## 原型链继承

|  |
| --- |
| *function* Person() {    this.name = '张三'    this.run = *function*() {        console.log(this.name + '...run')    }  }  Person.prototype.sex = '男'  Person.prototype.work = *function*(){      console.log(this.name + '...work')  }  // 原型链继承: 可以继承构造函数里的属性和实例方法 也可以继承原型链上的属性和方法  *function* Student() { }  Student.prototype = new Person()  *let* student = new Student()  student.run()  student.work() |

## 原型链继承问题

|  |
| --- |
| 问题：子类的时候没法给父类传参  *function* Person(*name*, *age*) {    this.name = *name*    this.age = *age*    this.run = *function*() {        console.log(this.name + '...run')    }  }  Person.prototype.sex = '男'  Person.prototype.work = *function*(){    console.log(this.name + '...work')  }  *function* Student(*name*,*age*) { }  // 原型链继承  Student.prototype = new Person()  // 实例化子类的时候没法给父类传参  *let* student = new Student('赵四', 20)  student.run() // undefined...run  student.work() // undefined...work |

## 原型链+对象冒充的组合继承

|  |
| --- |
| 原型链+对象冒充的组合继承：实例化子类的时可以给父类传参，也可以继承原型上的属性和方法  *function* Person(*name*, *age*){    this.name = *name*    this.age = *age*    this.run = *function*() {        console.log(this.name + '...run')    }  }  Person.prototype.sex = '男'  Person.prototype.work = *function*() {    console.log(this.name + '...work')  }  *function* Student(*name*,*age*) {    //对象冒充继承  可以继承构造函数里面的属性和方法、实例化子类可以给父类传参    Person.call(this, *name*, *age*)  }  Student.prototype = new Person()  // 实例化子类可以给父类传参,  *let* student = new Student('张三',20)  student.run() // 张三...run  // 可以继承原型链上的属性和方法  student.work() // 张三...work |

## 原型链+对象冒充继承的另一种方式

|  |
| --- |
| *function* Person(*name*, *age*) {    this.name = *name*    this.age = *age*    this.run = *function* () {      console.log(this.name + '...run')    }  }  Person.prototype.sex = '男'  Person.prototype.work = *function* () {    console.log(this.name + '...work')  }  *function* Student(*name*, *age*) {    //对象冒充继承  可以继承构造函数里面的属性和方法、实例化子类可以给父类传参    Person.call(this, *name*, *age*)  }  // 子类的原型指向父类的原型  Student.prototype = Person.prototype  *let* student = new Student('张三', 20)  student.run()  student.work() |

# typeScript类、继承

## 继承extends 、super

|  |
| --- |
| *class* Person{    name: *string*    //构造函数：实例化类时触发  *constructor*(*name*:*string*) {      this.name = *name*    }    run():*void* {      console.log(this.name, '...父类')    }  }  *let* p = new Person('王五')  p.run() // 王五 ...父类  *class* Student extends Person{  *constructor*(*name*:*string*){      // 初始化父类的构造函数      super(*name*)    }    // 子类和父类有相同方法时，子类调用自身的    run():*void* {      console.log(this.name, '...run子类')    }    work():*void* {      console.log(this.name, '...work子类')    }  }  *let* student = new Student('张三')  student.run() // 张三 ...run子类  student.work() // 张三 ...work子类 |

## 类的修饰符(访问权限)

|  |
| --- |
| public :公有 在当前类里面、 子类 、类外面都可以访问  protected：保护类型 在当前类里面、子类里面可以访问 ，在类外部没法访问  private ：私有 在当前类里面可以访问，子类、类外部都没法访问  属性如果不加修饰符 默认就是 公有 （public） |

### public

|  |
| --- |
| *class* Person {    public name:*string*  *constructor*(*name*:*string*){        this.name = *name*    }    run():*void* {      // 在父类里面可以访问      console.log(this.name, '...父类')    }  }  *let* p = new Person('王五')  p.run()  // 在父类外可以访问  console.log(p.name, '...父类')  *class* Student extends Person{  *constructor*(*name*:*string*){        super(*name*)    }    run():*void* {      // 在子类里面可以访问      console.log(this.name, '...run子类')    }  }  *let* student = new Student('张三')  student.run()  // 在子类外可以访问  console.log(student.name, '...run子类') |

### protected

|  |
| --- |
| *class* Person{    protected name:*string*  *constructor*(*name*:*string*){      this.name = *name*    }    run():*void* {      // 在父类里面可以访问      console.log(this.name, '...父类')    }  }  *let* p = new Person('王五')  p.run()  // 在父类外不可以访问  console.log(p.name, '...父类')  *class* Student extends Person{  *constructor*(*name*:*string*) {      super(*name*)    }    run():*void* {      // 在子类里面可以访问      console.log(this.name, '...run子类')    }  }  *let* student = new Student('张三')  student.run()  // 在子类外不可以访问  console.log(student.name, '...run子类') |

### private

|  |
| --- |
| *class* Person{    private name:*string*  *constructor*(*name*:*string*){      this.name = *name*    }    run():*void* {      // 在父类里面可以访问      console.log(this.name, '...父类')    }  }  *let* p = new Person('王五')  p.run()  // 在父类外不可以访问  console.log(p.name, '...父类')  *class* Student extends Person{  *constructor*(*name*:*string*) {      super(*name*)    }    run():*void* {      // 在子类里面不可以访问      console.log(this.name, '...run子类')    }  }  *let* student = new Student('张三')  student.run()  // 在子类外不可以访问  console.log(student.name, '...run子类') |

## 静态属性 静态方法

### ES5 静态属性 静态方法

|  |
| --- |
| *function* Person(*name*) {    // 实例属性    this.name1 = *name*    // 实例方法    this.run = *function*() {      console.log(this.name1 + '...实例方法')    }  }  // 静态属性  Person.name2 = '李四'  // 静态方法  Person.run = *function*() {    console.log(Person.name2 + '...静态方法')  }  *let* p = new Person('张三')  // 调用实例方法、实例属性  p.run() // 张三...实例方法  console.log(p.name1) // 张三  // 调用静态方法、静态属性  Person.run() // 李四...静态方法  console.log(Person.name2) // 李四 |

### typeScript静态属性 静态方法

|  |
| --- |
| *class* Person{    // 属性    public name:*string*    // 静态属性    static sex="男"  *constructor*(*name*:*string*) {      this.name = *name*    }    run():*void* {      console.log(this.name + '...run') // 张三...run      // 实例方法可以使用静态属性      console.log(Person.sex + '...run') // 男...run    }      static print() {      // 静态方法，不能直接调用类里面的属性      // console.log(this.name + '...print')      console.log(Person.sex+ '...print') // 男...print    }  }  *let* p:Person = new Person('张三')  p.run()  // 类名直接调用静态方法、静态属性  Person.print()  console.log(Person.sex) // 男 |

## 多态

|  |
| --- |
| // 多态: 父类定义一个方法不去实现，让继承它的子类去实现  每一个子类有不同的表现  // 多态属于继承  *class* Animal {    name:*string*  *constructor*(*name*:*string*) {        this.name= *name*    }    // 继承它的子类去实现 ，每个子类吃什么，不一样    eat() :*void*{        console.log('吃的方法')    }  }  *class* Dog extends Animal {  *constructor*(*name*:*string*){        super(*name*)    }    eat():*void* {       console.log(this.name + '吃狗粮')    }  }  *class* Cat extends Animal{  *constructor*(*name*:*string*) {        super(*name*)    }    eat():*void*{        console.log(this.name + '吃猫粮')    }  }  *let* dog = new Dog('狗')  dog.eat()  *let* cat = new Cat('猫')  cat.eat() |

## 抽象类

|  |
| --- |
| /\*  abstract关键字定义抽象类和抽象方法  1.抽象类是提供其他类继承的基类，不能直接被实例化  2.抽象方法只能放在抽象类里面  3.抽象方法不包含具体实现，且必须在派生类中实现  4.非抽象方法，派生类可以不实现  5.抽象类和抽象方法用来定义标准  \*/  //标准：Animal 这个类要求它的子类必须包含eat方法  abstract *class* Animal{    public name:*string*  *constructor*(*name*:*string*) {        this.name = *name*    }    // 抽象方法    abstract eat():*any*    // 非抽象方法    run(): *void*{      console.log('非抽象方法可以不实现')    }  }  // let animal = new Animal() // 错误  *class* Dog extends Animal {    *constructor*(*name*:*any*){        super(*name*)    }    // 抽象类的子类必须实现抽象类里的抽象方法    eat():*void* {      console.log(this.name + '吃狗粮')    }  }  *class* Cat extends Animal{  *constructor*(*name*:*any*){        super(*name*)    }    eat():*void* {      console.log(this.name + '吃猫粮')   }  }  *let* dog = new Dog('狗')  dog.eat()  *let* cat = new Cat('猫')  cat.eat() |

# 接口

|  |
| --- |
| 接口的作用：在面向对象的编程中，接口是一种规范的定义，它定义了行为和动作的规范，  在程序设计里面，接口起到一种限制和规范的作用。接口定义了某一批类所需要遵守的规范，  接口不关心这些类的内部状态数据，也不关心这些类里方法的实现细节，  它只规定这批类里必须提供某些方法，提供这些方法的类就可以满足实际需要。  typescrip中的接口类似于java，同时还增加了更灵活的接口类型，包括属性、函数、可索引和类等。  接口：行为和动作的规范  抽象类也是定义行为和动作的规范，但只适应于类，  接口对属性、函数、类等都可以定义行为和动作的规范 |

## 属性接口 (对json的约束)

|  |
| --- |
| // 自定义方法的参数, 对json进行约束  *function* printLabel(*labelInfo*:{label:*string*}):*void* {    console.log('printLabel')  }  printLabel('hahah')      // 错误写法  printLabel({name:'张三'})  // 错误的写法  printLabel({label:'张三'})  // 正确的写法  ====================================  // 对批量方法的参数(对象)进行约束  *interface* FullName{    //注意;结束    firstName:*string*;    secondName:*string*;  }  // 传入的参数必须包含 firstName  secondName  *function* printName(*name*:FullName) {    console.log(*name*.firstName+'--'+*name*.secondName)  }  // printName('10')  // 错误  --------------------------------------------------  // 定义变量obj传入，不报错  *let* obj = {    age:20,    firstName:'张',    secondName:'三'  }  printName(obj)  --------------------------------------------------  // 参数的顺序可以不一样  printName({      secondName:'secondName',      firstName:'firstName'  })  --------------------------------------------------  //  直接传入对象报错，只能传入firstName  secondName  printName({    age:20,    firstName:'张',    secondName:'三'  })    --------------------------------------------------  //  传入的参数必须包含 firstName  secondName  *function* printInfo(*info*:FullName) {    console.log(*info*.firstName+'--'+*info*.secondName)  }  printInfo({    firstName:'李',    secondName:'四'  }) |

## 可选属性

|  |
| --- |
| *interface* FullName{      firstName:*string*;      secondName?:*string*;  }  *function* getName(*name*:FullName){      console.log(*name*)  }  getName({      firstName:'firstName'  }) |

## 函数类型接口

|  |
| --- |
| 函数类型接口:对方法传入的参数，以及返回值进行约束 批量约束  // 加密的函数类型接口  *interface* encrypt {      (*key*:*string*,*value*:*string*):*string*;  }  *let* md5:encrypt = *function*(*key*:*string*,*value*:*string*):*string* {      //模拟操作      return *key* + *value*  }  console.log(md5('name','zhangsan'))  *let* sha1:encrypt=*function*(*key*:*string*,*value*:*string*):*string* {      //模拟操作      return *key* + *value*  }  console.log(sha1('name','lisi')) |

## 可索引接口

|  |
| --- |
|  |

## 类类型接口

|  |
| --- |
|  |

## 接口扩展

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |